```
/*Data Manipulation Using Correlated Subqueries*/


/*Subqueries are used heavily in update, delete, and insert statements
as well, with correlated subqueries appearing frequently in update
and delete statements.*/


desc rental;


UPDATE customer AS c
SET
    c.last_update = (SELECT
            MAX(r.rental_date)
        FROM
            rental r
        WHERE
            r.customer_id = c.customer_id)
WHERE
    EXISTS( SELECT
            1
        FROM
            rental r
        WHERE
            r.customer_id = c.customer_id);
```

```sql
/*DO NOT EXECUTE*/

DELETE FROM customer
WHERE
    365 < ALL (SELECT
        DATEDIFF(NOW(), r.rental_date) days_since_last_rental
    FROM
        rental r
    WHERE
        r.customer_id = customer.customer_id);

/*When using correlated subqueries with delete statements in MySQL,
keep in mind that, for whatever reason, table aliases are not allowed*/
```

```
/*When to Use Subqueries:

1. Subqueries as Data Sources

Since a subquery generates a result set containing rows and columns
of data, it is perfectly valid to include subqueries in your from
clause along with tables.*/


SELECT

    c.first_name, c.last_name, p.num_rentals, p.total_payments

FROM

    customer AS c

        INNER JOIN

    (SELECT

        customer_id,

            COUNT(*) AS num_rentals,

            SUM(amount) AS total_payments

    FROM

        payment

    GROUP BY customer_id) AS p ON c.customer_id = p.customer_id;


/*2. Data fabrication

Along with using subqueries to summarize existing data, you can use
subqueries to generate data that doesn’t exist in any form within your
database.

For example, you may wish to group your customers by the amount of
money spent on film rentals, but you want to use group definitions
that are not stored in your database.*/


SELECT 'Small Fry' name, 0 low_limit, 74.99 high_limit

UNION ALL SELECT 'Average Joes' name, 75 low_limit, 149.99 high_limit

UNION ALL SELECT 'Heavy Hitters' name, 150 low_limit, 9999999.99
high_limit;
```

```
SELECT
    pymnt_grps.name, COUNT(*) num_customers
FROM
    (SELECT
        customer_id, COUNT(*) num_rentals, SUM(amount) tot_payments
    FROM
        payment
    GROUP BY customer_id) pymnt
INNER JOIN
(SELECT 'Small Fry' name, 0 low_limit, 74.99 high_limit
UNION ALL
SELECT 'Average Joes' name, 75 low_limit, 149.99 high_limit
UNION ALL
SELECT 'Heavy Hitters' name, 150 low_limit, 9999999.99 high_limit
) pymnt_grps
ON pymnt.tot_payments
BETWEEN pymnt_grps.low_limit AND pymnt_grps.high_limit
GROUP BY pymnt_grps.name;
```

```
/*3. Task-oriented subqueries*/


SELECT

    c.first_name,

    c.last_name,

    ct.city,

    SUM(p.amount) AS total_payments,

    COUNT(*) AS total_rentals

FROM

    payment AS p

        INNER JOIN

    customer AS c ON p.customer_id = c.customer_id

        INNER JOIN

    address AS a ON c.address_id = a.address_id

        INNER JOIN

    city AS ct ON a.city_id = ct.city_id

GROUP BY c.first_name;


/*This query returns the desired data, but if you look at the query
closely, you will see that the customer, address, and city tables are
needed only for display purposes and that the payment table has
everything needed to generate the groupings (customer_id and
amount).*/
```

```
SELECT
    customer_id, COUNT(*) tot_rentals, SUM(amount) tot_payments
FROM
    payment
GROUP BY customer_id;

SELECT
    c.first_name,
    c.last_name,
    ct.city,
    pymnt.tot_payments,
    pymnt.tot_rentals
FROM
    (SELECT
        customer_id, COUNT(*) tot_rentals, SUM(amount) tot_payments
    FROM
        payment
    GROUP BY customer_id) AS pymnt
        INNER JOIN
    customer c ON pymnt.customer_id = c.customer_id
        INNER JOIN
    address a ON c.address_id = a.address_id
        INNER JOIN
    city ct ON a.city_id = ct.city_id;
```

```
/*Common table expressions (CTE)*/

WITH actors_s AS
(SELECT actor_id, first_name, last_name
FROM actor
WHERE last_name LIKE 'S%'
),

actors_s_pg AS
(SELECT s.actor_id, s.first_name, s.last_name,
f.film_id, f.title
FROM actors_s s
INNER JOIN film_actor fa
ON s.actor_id = fa.actor_id
INNER JOIN film f
ON f.film_id = fa.film_id
WHERE f.rating = 'PG'
),

actors_s_pg_revenue AS
(SELECT spg.first_name, spg.last_name, p.amount
FROM actors_s_pg spg
INNER JOIN inventory i
ON i.film_id = spg.film_id
INNER JOIN rental r
ON i.inventory_id = r.inventory_id
INNER JOIN payment p
ON r.rental_id = p.rental_id
) -- end of With clause
```

```
SELECT spg_rev.first_name, spg_rev.last_name,
sum(spg_rev.amount) tot_revenue
FROM actors_s_pg_revenue spg_rev
GROUP BY spg_rev.first_name, spg_rev.last_name
ORDER BY 3 desc;
```

/***Subqueries as Expression Generators***/

```
SELECT
    (SELECT
            c.first_name
        FROM
            customer c
        WHERE
            c.customer_id = p.customer_id) first_name,
    (SELECT
            c.last_name
        FROM
            customer c
        WHERE
            c.customer_id = p.customer_id) last_name,
    (SELECT
            ct.city
        FROM
            customer c
                INNER JOIN
            address a ON c.address_id = a.address_id
                INNER JOIN
            city ct ON a.city_id = ct.city_id
        WHERE
```

```
            c.customer_id = p.customer_id) city,
    SUM(p.amount) tot_payments,
    COUNT(*) tot_rentals
FROM
    payment p
GROUP BY p.customer_id;
```

/*There are two main differences between this query and the earlier version using a subquery in the from clause:

• Instead of joining the customer, address, and city tables to the payment data, correlated scalar subqueries are used in the select clause to look up the customer's first/last names and city.

• The customer table is accessed three times (once in each of the three subqueries) rather than just once.

As previously noted, scalar subqueries can also appear in the order by clause.*/

```
SELECT
    a.actor_id, a.first_name, a.last_name
FROM
    actor a
ORDER BY (SELECT
        COUNT(*)
    FROM
        film_actor fa
    WHERE
        fa.actor_id = a.actor_id) DESC;
```

```sql
/*Along with using correlated scalar subqueries in select statements,
you can use noncorrelated scalar subqueries to generate values for an
insert statement.*/


INSERT INTO film_actor (actor_id, film_id, last_update)
VALUES (
(SELECT actor_id FROM actor
WHERE first_name = 'JENNIFER' AND last_name = 'DAVIS'),
(SELECT film_id FROM film
WHERE title = 'ACE GOLDFINGER'),
now()
);
```

```
/*Exercise - 1
Construct a query against the film table that uses a filter condition
with a noncorrelated
subquery  against  the  category  table  to  find  all  action  films
(category.name =
'Action').*/


desc film;


desc category;


desc film_category;


SELECT
    title
FROM
    film
WHERE
    film_id IN (SELECT
            fc.film_id
        FROM
            film_category AS fc
                INNER JOIN
            category AS c ON fc.category_id = c.category_id
        WHERE
            c.name = 'Action');
```

```
/*Exercise - 2

Rework the query from Exercise 9-1 using a correlated subquery against
the category and film_category tables to achieve the same results.*/


SELECT

    f.title

FROM

    film AS f

WHERE

    EXISTS( SELECT

            1

        FROM

            film_category AS fc

                INNER JOIN

            category AS c ON fc.category_id = c.category_id

        WHERE

            c.name = 'Action'

                AND fc.film_id = f.film_id);
```

```
/*Exercise - 3

Join the following query to a subquery against the film_actor table
to show the level of each actor:

SELECT 'Hollywood Star' level, 30 min_roles, 99999 max_roles

UNION ALL

SELECT 'Prolific Actor' level, 20 min_roles, 29 max_roles

UNION ALL

SELECT 'Newcomer' level, 1 min_roles, 19 max_roles

The subquery against the film_actor table should count the number of
rows for each actor using group by actor_id, and the count should be
compared to the min_roles/max_roles columns to determine which level
each actor belongs to.*/


SELECT

    actr.actor_id, grps.level

FROM

    (SELECT

        actor_id, COUNT(*) num_roles

    FROM

        film_actor

    GROUP BY actor_id) actr

        INNER JOIN

    (SELECT 'Hollywood Star' level, 30 min_roles, 99999 max_roles

    UNION ALL

    SELECT 'Prolific Actor' level, 20 min_roles, 29 max_roles

    UNION ALL

    SELECT 'Newcomer' level, 1 min_roles, 19 max_roles) grps

    ON actr.num_roles

    BETWEEN grps.min_roles AND grps.max_roles;
```